

Speeding up SOR Solvers for Constraint-based GUIs with a Warm-Start Strategy

Noreen Jamil, Johannes Müller, Christof Lutteroth and Gerald Weber

Department of Computer Science
University of Auckland

Private Bag 92019, Auckland, New Zealand

{njam031, jmue933}@aucklanduni.ac.nz, {lutteroth, gerald}@cs.auckland.ac.nz

Abstract—Many computer programs have graphical user interfaces (GUIs), which need good layout to make efficient use of the available screen real estate. Most GUIs do not have a fixed layout, but are resizable and able to adapt themselves. Constraints are a powerful tool for specifying adaptable GUI layouts: they are used to specify a layout in a general form, and a constraint solver is used to find a satisfying concrete layout, e.g. for a specific GUI size. The constraint solver has to calculate a new layout every time a GUI is resized or changed, so it needs to be efficient to ensure a good user experience. One approach for constraint solvers is based on the Gauss-Seidel algorithm and successive over-relaxation (SOR).

Our observation is that a solution after resizing or changing is similar in structure to a previous solution. Thus, our hypothesis is that we can increase the computational performance of an SOR-based constraint solver if we reuse the solution of a previous layout to warm-start the solving of a new layout. In this paper we report on experiments to test this hypothesis experimentally for three common use cases: big-step resizing, small-step resizing and constraint change. In our experiments, we measured the solving time for randomly generated GUI layout specifications of various sizes. For all three cases we found that the performance is improved if an existing solution is used as a starting solution for a new layout.

Index Terms—UI layout, warm start, successive over-relaxation.

I. INTRODUCTION

Various numerical methods have been introduced to solve linear problems as they appear in engineering, mathematics and computer science. These methods can be divided into direct and iterative methods. Direct methods aim to calculate an exact solution in a finite number of operations, whereas iterative methods begin with an initial approximation and usually produce improved approximations in a theoretically infinite sequence whose limit is the exact solution [1].

Many linear problems are sparse, i.e. most linear coefficients in the corresponding matrix are zero so that the number of non-zero coefficients is $O(n)$ with n being the number of variables [2]. Iterative methods do not spend processing time on coefficients that are zero. Direct methods, in contrast, usually lead to fill-in, i.e. coefficients change from an initial zero to a non-zero value during the execution of the algorithm. In these methods we therefore may weaken the sparsity property and may have to deal with more coefficients, which makes the processing time slower. Therefore, iterative

methods are often faster than naive direct methods in such cases. In this paper, we are concerned with a domain where sparse problems occur frequently, namely constraint-based graphical user interface (GUI) layout. Constraint-based layout models have been studied for quite a while in the research community [3]–[6] and attracted attention recently because of a newly introduced layout model in the Cocoa API of Apple’s Mac OS X¹.

A common iterative method is successive over-relaxation (SOR) [7]. Starting with an initial guess, it repeatedly iterates through the constraints of a linear specification, refining the solution until a sufficient precision is reached. For each constraint it chooses a *pivot variable*, and changes the value of that variable so that the constraint is satisfied. In order to use SOR for GUI layout, certain considerations and extensions are necessary.

First, linear problems in GUI layout are often over-determined and have many inequality constraints, leading to non-square coefficient matrices. This leads to the problem of *pivot assignment*: this is the problem of choosing a pivot variable for each constraint so that the iterative method converges. The standard SOR algorithms choose for each constraint the pivot variable on the diagonal of the problem matrix. In over-determined systems we do not have a main diagonal, therefore we make use of the pivot assignment algorithms proposed in [8].

Second, constraint-based GUI specifications may contain conflicting constraints, rendering a specification infeasible. To deal with conflicts, *soft constraints* need to be supported. In contrast to the usual *hard constraints*, which cannot be violated, soft constraints may be violated as much as necessary if no other solution can be found. Soft constraints can be prioritized so that in a conflict between two soft constraints only the soft constraint with the lower priority is violated [9]. Using only soft constraints has the advantage that a problem is always solvable, which cannot be guaranteed if hard constraints are used. In this work, we use the soft constraint algorithms proposed in [8].

A GUI layout specification has to be solved whenever the conditions under which a GUI is displayed or the GUI itself change. Most GUIs can be resized, e.g. to adapt to different

¹Cocoa Auto Layout Guide, 2012, <http://developer.apple.com>

screen sizes or to let the user choose an appropriate size dynamically. Sometimes GUIs need to be changed dynamically to adjust to content of different sizes. Each time a GUI is resized or changed, the existing GUI layout specification is changed and a new specification is created. However, the new specification is similar to the previous one because the widgets and their relations typically stay the same. Usually, only some size parameters change. For example, Figure 1 shows a GUI that is resized, with the corresponding constraint specifications. Only the height constraint at the beginning of the specification is changed.

As a consequence, constraint solvers for GUI layout usually have to solve specifications that are similar to the specification that has been solved previously. For that reason, it seems plausible that the previous solution is a good initial value for the iterative solving process – something that is known as a warm-start strategy. This leads us to the following hypothesis: *Using a previous solution of a GUI constraint specification to warm-start an SOR solver reduces the solving time.*

We tested the hypothesis by considering three common use cases where GUIs are changed during runtime. First, small-step resizing, where the GUI size is changed by a small amount, e.g. when it is resized by a user dynamically. Second, big-step resizing, where the GUI size is changed by a larger amount, e.g. when the GUI size is maximized. And third, changes of several constraints, e.g. when the sizes of labels are adjusted for a different language. The solving time when using SOR with and without a warm-start strategy was compared for the three use cases, using randomly generated layout specifications of different sizes.

Section II sums up related work about constraint solving and warm-start strategies. Section III puts this research into context with background information about constraint-based GUI-layout, SOR, pivot assignment and conflict resolution. Section IV presents the methodology and results of our performance experiment. Section V concludes the paper.

II. RELATED WORK

The overall problem, solving linear systems for constraint-based GUIs, is related to solution procedures for over-determined linear systems in general and constraint-based GUIs in particular. Several direct and iterative methods exist, which can solve over-determined systems in a least-square sense [10]. Examples are QR-factorization [10], the simplex algorithm [11], the conjugate gradient method [12] and the GMRES-method [12]. They are the basis for solvers specifically designed to solve problems of constraint-based GUIs. Some are based on direct methods, for example HiRise and HiRise2 [13] but the vast majority of existing solvers is based on convex optimization approaches and uses slack variables and an objective function [3], [4], [14]. These methods can handle simultaneous constraints, i.e. constraints that depend on each other. In that respect they are superior to local propagation algorithms, such as DeltaBlue [15] and SkyBlue [16], which cannot do so.

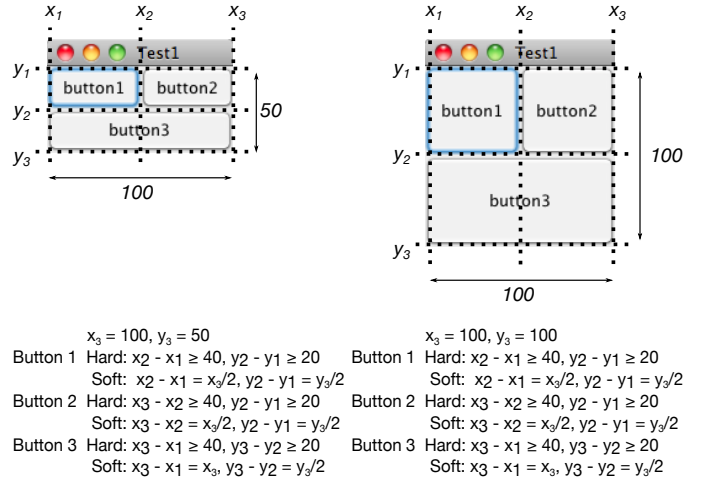


Fig. 1. A GUI constraint specification before and after resizing

Approaches related to warm-start strategies have been proposed in numerous previous works. Lessard [17] analyzed computational speed and the effect of warm-starting for different iterative methods with large systems, including the multigrid method, the preconditioned conjugate-gradient method, and several new variants of these methods. Using a previous estimate for initializing an iterative scheme could reduce computation time significantly. Wright et al. [18] used a warm-start strategy to speed up gradient projection for sparse reconstruction (GPSR) and iterative shrinkage/thresholding (IST) algorithms. Other methods for accelerating convergence by using warm-start techniques in iterative solution procedures are described in [19]–[21]. The use of warm-start strategies for constraint-based GUI layout problems using SOR has not been explored before.

III. BACKGROUND

To put our study into context, this section gives a short overview of constraint-based GUIs, SOR, and the extensions of SOR necessary to solve GUI layout specifications.

A. User Interface Layout as a Linear Problem

Constraints are a suitable mechanism for specifying the relationships among objects. They are used in the area of logic programming, artificial intelligence and GUI specification. They can be used to describe problems that are difficult to solve, conveniently decoupling the description of the problems from their solution. Due to this property, constraints are a suitable way of specifying GUI layouts, where the objects are widgets and the relationships between them are spatial relationships such as alignment and proportions. In addition to the relationships to other widgets, each widget has its own set of constraints describing properties such as minimum, maximum and preferred size.

GUI layouts are often specified with linear constraints [4]. The positions and sizes of the widgets in a layout translate to variables. Constraints about alignment and proportions

translate to linear equations, and constraints about minimum and maximum sizes translate to linear inequalities. There are constraints for each widget that relate each of its four boundaries to another part of the layout, or specify boundary values for the widget's size, as shown in Figure 1. As a result, the direct interaction between constraints is limited by the topology of a layout, resulting in sparsity of the linear specification.

During application runtime, GUIs need to be adapted to changing conditions such as the available GUI size. This is done by changing some of the constraints, typically a small number. For example, the overall size of the GUI is typically specified in by constraints: one for the width and one for the height. When the GUI size changes, only these two constraints need to be adjusted, as shown in Figure 1. Another typical situation where constraints need to be changed is when preferred sizes change. For example, if the language settings are changed in an application, the preferred sizes of textual labels have to adjust to the new language.

B. Successive Over-Relaxation (SOR)

Most of the research on iterative methods deals with iterative methods for solving linear systems of equalities and inequalities for sparse square matrices, the most important method being SOR, also known as linear relaxation. This section summarizes the most important findings.

The best-known iterative method for solving linear constraints is the Gauss-Seidel method [22]. Given a system of n equations and n variables of the form

$$Ax = b \quad (1)$$

we can rewrite the equation for the i th term as follows

$$x_i = \frac{1}{a_{ii}}(b_i - \sum_{j=1}^n a_{ij}x_j). \quad (2)$$

The variable x_i , which is brought onto the left side, is called the *pivot variable*, and a_{ii} is the *pivot coefficient* or *pivot element* chosen for row i . An initial estimate for x is chosen, which usually does not fulfill the equations. The algorithm refines the estimate by repeatedly replacing all individual x_i so that the i th equation becomes fulfilled. This is done in round-robin fashion, and one full run through all n equations is one iteration, r being the iteration number. We can therefore write the process as

$$x_i^{r+1} = \frac{1}{a_{ii}}(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{r+1} - \sum_{j=i+1}^n a_{ij}x_j^r). \quad (3)$$

The algorithm iterates until the relative approximate error is less than a pre-specified tolerance.

SOR, also known as linear relaxation, is an improvement of the Gauss-Seidel method [7]. It is used to speed up the convergence of the Gauss-Seidel method by introducing a

parameter ω , known as relaxation parameter, so that

$$x_i^{r+1} = \frac{\omega}{a_{ii}}(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{r+1} - \sum_{j=i+1}^n a_{ij}x_j^r) + (1 - \omega)x_i^r. \quad (4)$$

This reduces to the Gauss-Seidel method if $\omega = 1$. It is known as over-relaxation if $\omega > 1$, and known as under-relaxation if $\omega < 1$.

Definition 1. *The spectral radius of a matrix is the maximum of the absolute values of its eigenvalues.*

The convergence rate of the SOR method depends on the spectral radius of the coefficient matrix of the problem. The smaller the spectral radius is, the faster the SOR method converges. We usually have well-conditioned coefficient matrices in the GUI layout domain for which the SOR method converges.

SOR supports linear equalities as well as inequalities. Inequalities are handled similar to equalities [23], [24]: in each iteration, inequalities are ignored if they are satisfied, and otherwise treated as if they were equalities.

C. Pivot Assignment

In the case of square coefficient matrices, pivots are selected on the main diagonal. This is not possible in the over-determined case. Since the diagonal elements do not lend themselves naturally as pivot elements if the matrix is non-square, we need to explicitly select a pivot element for each constraint. In other words, we need to determine a *pivot assignment*.

Definition 2. *A pivot assignment is an assignment of constraints to variables.*

$$\gamma: \text{Constraints} \rightarrow \text{Variables}$$

In previous work, we suggested algorithms which select a pivot element randomly or according to some criteria [8]. We use our random pivot assignment algorithm for the study presented in this paper. This algorithm assigns the pivot variable for each constraint randomly in each iteration, so that the assignment varies over the iterations.

It is not inherently obvious that randomized assignments work for the linear relaxation approach, but it is the simplest approach that may work. Although the random algorithm does generally not make the optimal assignment with regard to convergence, it reduces the effect of bad assignments while allowing for good assignments. In particular, it is guaranteed that every suitable variable will be chosen as pivot variable at some point. The general assumption underlying randomized algorithms is that the effect of good choices outweighs the effect of bad choices.

In the general case constraint-based GUIs are over-determined, which can result in conflicts between constraints of the problem. A proper pivot assignment algorithm alone is not sufficient to deal with such cases. A technique to handle conflicts between constraints, e.g. in the form of soft constraints, is required.

D. Conflict Resolution

To resolve conflicts in over-determined systems, soft constraints are introduced. A natural way to support soft constraints is to assign priorities to all constraints. These priorities can be defined as a total order on all constraints that specifies which one of two constraints should be violated in case of a conflict.

In our study we use the constraint insertion algorithm proposed in previous work [8]. This algorithm tests constraints incrementally. We start with an empty set E of enabled constraints. Iterating through the constraints in order of descending priority, we add each constraint tentatively to E (“enabling” it), and try to solve the resulting specification. If a solution is found, then we proceed to the next constraint. If no solution is found within a fixed maximum number of iterations, then the tentatively added constraint is removed again. In that case, the previous solution is restored and we proceed to the next constraint.

IV. EXPERIMENT

In this section, we evaluate the use of warm-starting for GUI layout problems using SOR. We tested specifically the effect of warm-starting a constraint solver on the performance in terms of computation time.

A. Methodology

We conducted the experiments with our implementation of an SOR solver for GUI layout, which uses random pivot assignment and constraint insertion as a conflict resolution strategy. We used two versions of that solver: the first version started every solver run with an initial solution $x = (0, \dots, 0)$, the second version started every solver run with the optimal solution from the previous run $x = x_{\text{prev}}^*$.

We evaluated the following three use cases:

- 1) *Small-step resizing*: The width and height of the window was randomly changed by a value in between 0 and 3 pixels.
- 2) *Big-step resizing*: The width and height of the GUI window was randomly changed by a value between 4 and 3000 pixels.
- 3) *Constraint change*: 10 per cent of all constraints of a GUI were randomly changed.

Small-step resizing occurs in practice when a window is continuously resized by dragging a window border. Big-step resizing occurs when a GUI is initially loaded on different screens, when a GUI is switched to or from full-screen mode, or when the orientation of a screen is changed. Constraint changes as in use case 3 occur, for example, when several preferred sizes change as a result of changing the language of an application.

Layout specifications were randomly generated using the parameterized algorithm described in [4]. The problem size was varied from 0 to 201 areas. For each area 4 constraints are added, which specify the position of the area in the layout. Additionally, a specification needs 4 constraints to define the size of the window. So we started with a problem of 4

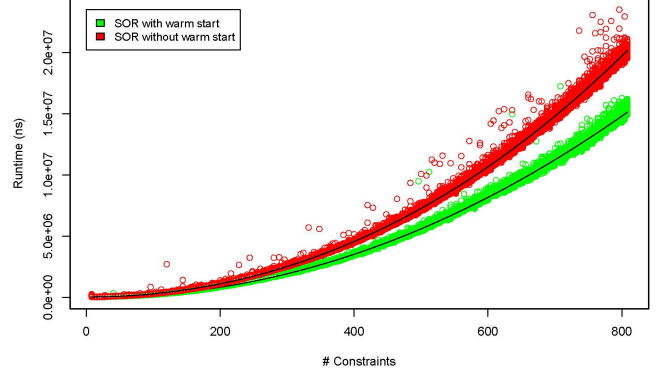


Fig. 2. Small-step resizing performance results

constraints and ended with a problem of 808 constraints. For each size, 10 random layouts were evaluated. For each of the three use cases, each of these random layouts was changed 20 times, and the solving time was measured. A linear relaxation parameter of 0.7 and a tolerance of 0.01 were used for solving. The measurements were performed on a desktop computer with Intel i5 3.3GHz processor and 64-bit Windows 7 running an Oracle Java virtual machine.

Symbol	Explanation
β_0	Intercept of the regression model
β_{1-3}	Estimated model parameters
c	Number of constraints
T	Measured time in milliseconds
R^2	Coefficient of determination

TABLE I
SYMBOLS

To compare the performance of both versions of the solver we used a regression model

$$T = f(c) + \epsilon$$

and examined the estimated model visually and numerically. See Table I for an explanation of the symbols used.

B. Results

To identify the performance trend of the solvers, we tried different regression functions f (linear, quadratic, log, cubic). We found that the best fitting model is the polynomial model

$$T = \beta_0 + \beta_1 c + \beta_2 c^2 + \beta_3 c^3 + \epsilon.$$

Key parameters of the regression models are depicted in Table II. A graphical representation of the measurements and the models can be found in Figures 2, 3 and 4. The results suggest a better performance of the solver with the warm-start strategy for all three use cases.

The variance of the measured runtime differs noticeably for both approaches. It is smaller for the rather small changes in

TABLE II
REGRESSION MODELS FOR SOLVERS WITH AND WITHOUT WARM-START STRATEGY

Strategy	β_0	β_1	β_2	β_3	R^2
Small-step resizing with warm start	$6.508 \cdot 10^4$ ***	$-8.940 \cdot 10^2$ ***	23.55***	$7.576 \cdot 10^{-4}$ ***	0.999
Small-step resizing without warm start	$4.104 \cdot 10^4$ ***	68.00***	26.05***	$5.971 \cdot 10^{-3}$ ***	0.999
Big-step resizing with warm start	$3.186 \cdot 10^4$ ***	-89.32 ***	13.42***	$3.202 \cdot 10^{-3}$ ***	0.996
Big-step resizing without warm start	$1.208 \cdot 10^4$ ***	$3.729 \cdot 10^2$ ***	18.24***	$4.921 \cdot 10^{-3}$ ***	0.999
Constraint changes with warm start	$1.074 \cdot 10^5$ ***	$-1.289 \cdot 10^3$ ***	21.88***	$-2.721 \cdot 10^{-4}$ ***	0.976
Constraint changes without warm start	$1.231 \cdot 10^5$ ***	$-1.648 \cdot 10^3$ ***	30.17***	$-2.837 \cdot 10^{-4}$ ***	0.962

Significance codes: *** $p < 0.001$

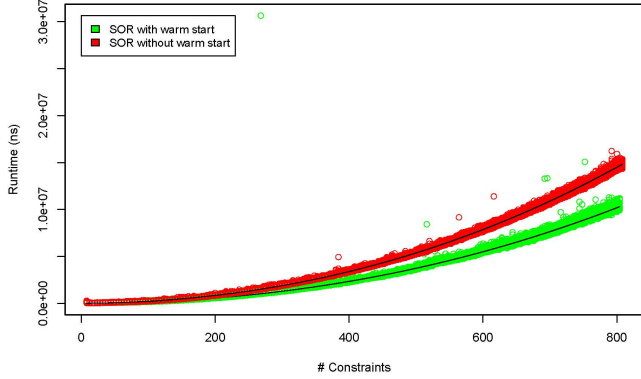


Fig. 3. Big-step resizing performance results

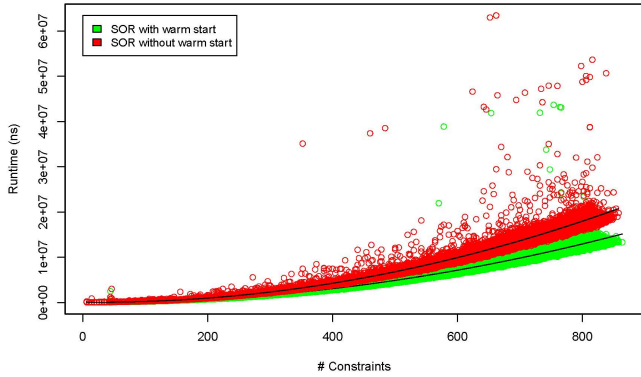


Fig. 4. Constraint change performance results

small-step resizing, and bigger for the big-step resizing and constraint change use cases. Especially for constraint change, this indicates that some problems with a lot of conflicts were generated, which require more iterations and hence a longer runtime. The measurements indicate that the variance of the runtime with warm start is smaller than without warm start, and this might be worth further analysis. It is somewhat astonishing that the experiments reveal only a relatively small effect of the warm-start strategy. One reason could be the use of the random pivot selector. Since it selects pivot elements

randomly, it can select pivot elements which let the solution deviate strongly from the initial solution before it actually converges towards the new solution. Another reason can be that the changes in the specification – even though they are fairly small – drastically change the solution in some cases. This can be, for example, due to conflict resolution. Some constraints, which were not satisfied with the old specification, can become satisfiable and suddenly have an effect on the solution after the specification was changed. Similarly, small changes in the specification can lead to new conflicts and hence disabling of constraints.

The effects of the warm-start strategy are comparable for all three use cases, but are the strongest for small-step resizing. This is convenient, as speed is of particular importance for the small-step resizing use case. Small-step resizing is typically done interactively by the user, and for a good user experience the GUI should react to such resizing in real-time.

V. CONCLUSION

In constraint-based GUIs with dynamic behavior, the specification that represents the layout of the GUI is often changed, e.g. when a window is resized. These changes are usually small, resulting in specifications that are very similar. Since the specifications are similar, one can expect also the results to be similar. Therefore, we evaluated the use of a warm-start strategy to improve the efficiency of SOR-based constraint solvers for GUIs. Three common use cases were evaluated with randomly generated GUI layouts: small-step resizing, big-step resizing, and random changes of several constraints.

We found that an SOR-based solver with a warm-start strategy indeed exhibits a better runtime behavior than a solver without warm-start strategy. Implementing a warm-start strategy in such solvers does not introduce additional computational effort, as existing values are simply reused. It is therefore advisable to equip SOR-based GUI layout solvers with a warm-start strategy.

However, we also found that the effect of a warm-start strategy is weaker than we expected. Possible reasons are the random pivot assignment and the constraint insertion conflict resolution strategy used in the experiment. Thus, a future work would be to explore the effect of warm starts also for other types of pivot selectors and conflict resolution strategies.

REFERENCES

- [1] Y. Saad, *Iterative methods for sparse linear systems*. SIAM, 2003.
- [2] S. Kunis and H. Rauhut, "Random sampling of sparse trigonometric polynomials, ii. orthogonal matching pursuit versus basis pursuit," *Journal Foundations of Computational Mathematics*, vol. 8, no. 6, pp. 737–763, Nov. 2008.
- [3] A. Borning, K. Marriott, P. Stuckey, and Y. Xiao, "Solving linear arithmetic constraints for user interface applications," in *Proceedings of the 10th annual ACM symposium on User interface software and technology (UIST '97)*. ACM, 1997, pp. 87–96.
- [4] C. Lutteroth, R. Strandh, and G. Weber, "Domain specific High-Level constraints for user interface layout," *Constraints*, vol. 13, no. 3, 2008.
- [5] A. Scoditti and W. Stuerzlinger, "A new layout method for graphical user interfaces," in *Science and Technology for Humanity (TIC-STH), 2009 IEEE Toronto International Conference*. IEEE, 2009, pp. 642–647.
- [6] C. Zeidler, C. Lutteroth, and G. Weber, "Constraint solving for beautiful user interfaces: how solving strategies support layout aesthetics," in *Proceedings of the 13th International Conference of the NZ Chapter of the ACM's Special Interest Group on Human-Computer Interaction*, ser. CHINZ '12. ACM, 2012, pp. 72–79.
- [7] D. M. Young, *Iterative Solution of Large Linear Systems*. Academic Press, 1971.
- [8] N. Jamil, J. Müller, C. Lutteroth, and G. Weber, "Extending linear relaxation for user interface layout," *Proceedings of 24th International Conference on Tools with Artificial Intelligence (ICTAI)*, 2012.
- [9] A. Borning, B. Freeman-Benson, and M. Wilson, "Constraint hierarchies," *Lisp and Symbolic Computation*, vol. 5, no. 3, pp. 223–270, 1992.
- [10] A. Björck, *Numerical Methods for Least Squares Problems*, ser. Handbook of Numerical Analysis. Society for Industrial and Applied Mathematics, 1996.
- [11] G. B. Dantzig, *Linear Programming and Extensions*, 11st ed., ser. Princeton Landmarks in Mathematics. Princeton Uni. Press, 1998.
- [12] G. Golub and C. Van Loan, *Matrix Computations*. Johns Hopkins Uni. Press, 1996.
- [13] H. Hosobe, "A simplex-based scalable linear constraint solver for user interface applications," in *Tools with Artificial Intelligence (ICTAI), 2011 23rd IEEE International Conference on*, Nov. 2011, pp. 793–798.
- [14] G. J. Badros, A. Borning, and P. J. Stuckey, "The cassowary linear arithmetic constraint solving algorithm," *ACM Transactions on Computer-Human Interaction*, vol. 8, no. 4, pp. 267–306, 2001.
- [15] J. M. Freeman-Benson and A. Borning, "An incremental constraint solver," *Communications of the ACM*, vol. 33, no. 1, pp. 54–63, 1990.
- [16] M. Sannella, "Skyblue: a multi-way local propagation constraint solver for user interface construction," in *Proceedings of the 7th annual ACM symposium on User interface software and technology (UIST '94)*. ACM, 1994, pp. 137–146.
- [17] L. Lessard, M. West, D. MacMynowski, and S. Lall, "Warm-started wavefront reconstruction for adaptive optics," *J. Opt. Soc. Am. A*, vol. 25, no. 5, pp. 1147–1155, May 2008.
- [18] S. Wright, R. Nowak, and M. A. T. Figueiredo, "Sparse reconstruction by separable approximation," *Signal Processing, IEEE Transactions on*, vol. 57, no. 7, pp. 2479–2493, 2009.
- [19] X. Ren and Z. Lin, "Linearized alternating direction method with adaptive penalty and warm starts for fast solving transform invariant low-rank textures," *International Journal of Computer Vision*, pp. 1–14, 2013.
- [20] A. Forsgren, "On warm starts for interior methods," in *System Modelling and Optimization*, 2005, pp. 51–66.
- [21] R. Fletcher, *Practical methods of optimization; (2nd ed.)*. Wiley-Interscience, 1987.
- [22] A. B. Saeed and A. B. Naeem, *Numerical Analysis*. Shahryar, 2008.
- [23] S. Agmon, "The relaxation method for linear inequalities," *Canadian Journal of Mathematics*, pp. 382–392, 1954.
- [24] T. Motzkin and I. Schoenberg, "The relaxation method for linear inequalities," *Canadian Journal of Mathematics*, pp. 393–404, 1954.